What is a Higher Order Component?

A Higher Order Component (HOC) is a function that takes a component as an input and returns a new component with additional functionality. It enhances the capabilities of the original component without modifying its code.

Use Cases for Higher Order Components

- Reusability: HOCs enable code reuse by encapsulating common functionality in a separate component.
- Authentication: HOCs can handle authentication logic and restrict access to certain components.
- Logging: HOCs can log component lifecycle events or actions for debugging purposes.

Implementing a Higher Order Component

```
const withLogger = (WrappedComponent) => {
  class WithLogger extends React.Component {
    componentDidMount() {
      console.log(`Component ${WrappedComponent.name} mounted`);
    }

    render() {
      return <WrappedComponent {...this.props} />;
    }
  }

  return WithLogger;
};

const MyComponent = (props) => {
  // Component logic here
};

const EnhancedComponent = withLogger(MyComponent);
```

Understanding Props

Props (short for properties) are used to pass data from a parent component to its child component. They are read-only and cannot be modified by the child component.

```
const ParentComponent = () => {
  const message = "Hello, React!";

  return <ChildComponent message={message} />;
};

const ChildComponent = (props) => {
  return <p>{props.message}</p>;
};
```

Default Props and Prop Types

- Default Props: You can define default values for props in case they are not provided.
- Prop Types: PropTypes allow you to validate the types of props being passed to a component, ensuring data integrity.

```
import PropTypes from 'prop-types';

const MyComponent = (props) => {
  // Component logic here
};

MyComponent.defaultProps = {
  count: 0,
};

MyComponent.propTypes = {
  count: PropTypes.number.isRequired,
};
```

What is Prop Drilling?

Prop drilling refers to the process of passing props through multiple levels of nested components to reach a component that needs the data. It can lead to cluttered and less maintainable code.

Avoiding Prop Drilling

- Context API: Use the Context API to provide data to components deep in the component tree without explicitly passing props through each intermediate component.
- Redux: Redux is a popular state management library that can eliminate prop drilling by providing a central store accessible by all components.

Example of Prop Drilling

```
const ParentComponent = () => {
  const message = "Hello, React!";

  return <ChildComponent message={message} />;
};

const ChildComponent = (props) => {
  return <GrandChildComponent message={props.message} />;
};

const GrandChildComponent = (props) => {
  return <p>{props.message}</p>;
};
```

Overview of Component Lifecycle

The React component lifecycle consists of different phases that a component goes through, from initialization to unmounting. Each phase has specific lifecycle methods associated with it.

Diagram - Component Lifecycle

```
          Initialization
                |
          Mounting Phase
                |
          Updating Phase
                |
         Unmounting Phase
```

These methods allow you to perform specific actions when the component is created, updated, or destroyed. Let's go through the different lifecycle methods with code snippets and examples:

Here's the correct order of the most commonly used lifecycle methods:

- constructor(props)
- static getDerivedStateFromProps(props, state)
- render()
- componentDidMount()
- componentDidUpdate(prevProps, prevState)
- shouldComponentUpdate(nextProps, nextState)
- componentWillUnmount()

Different Lifecycle methods

1. constructor(props): The constructor is the first method called when a component is created. It is used to initialize state and bind event handlers. Here's an example:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return <div>{this.state.count}</div>;
  }
}
```

2. componentDidMount(): The componentDidMount method is invoked immediately after the component is mounted (inserted into the DOM tree). It is commonly used to make API calls or initialize third-party libraries. Here's an example:

```
class MyComponent extends React.Component {
  componentDidMount() {
    // Make an API call or initialize a library
    fetchData().then(data => {
      this.setState({ data });
    });
  }
  render() {
    return <div>{this.state.data}</div>;
  }
}
```

3. componentDidUpdate(prevProps, prevState): The componentDidUpdate method is called after the component is updated. It is useful for performing side effects when props or state change. Make sure to include a condition to prevent an infinite loop when updating state inside this method. Here's an example:

```
class MyComponent extends React.Component {
  componentDidUpdate(prevProps, prevState) {
    if (this.state.count !== prevState.count) {
      console.log('Count updated:', this.state.count);
    }
  }

  render() {
    return <div>{this.state.count}</div>;
  }
}
```

4. shouldComponentUpdate(nextProps, nextState): The shouldComponentUpdate method is invoked before the component is updated. It allows you to control whether the component should re-render or not. Returning false from this method will prevent the component from updating. Here's an example:

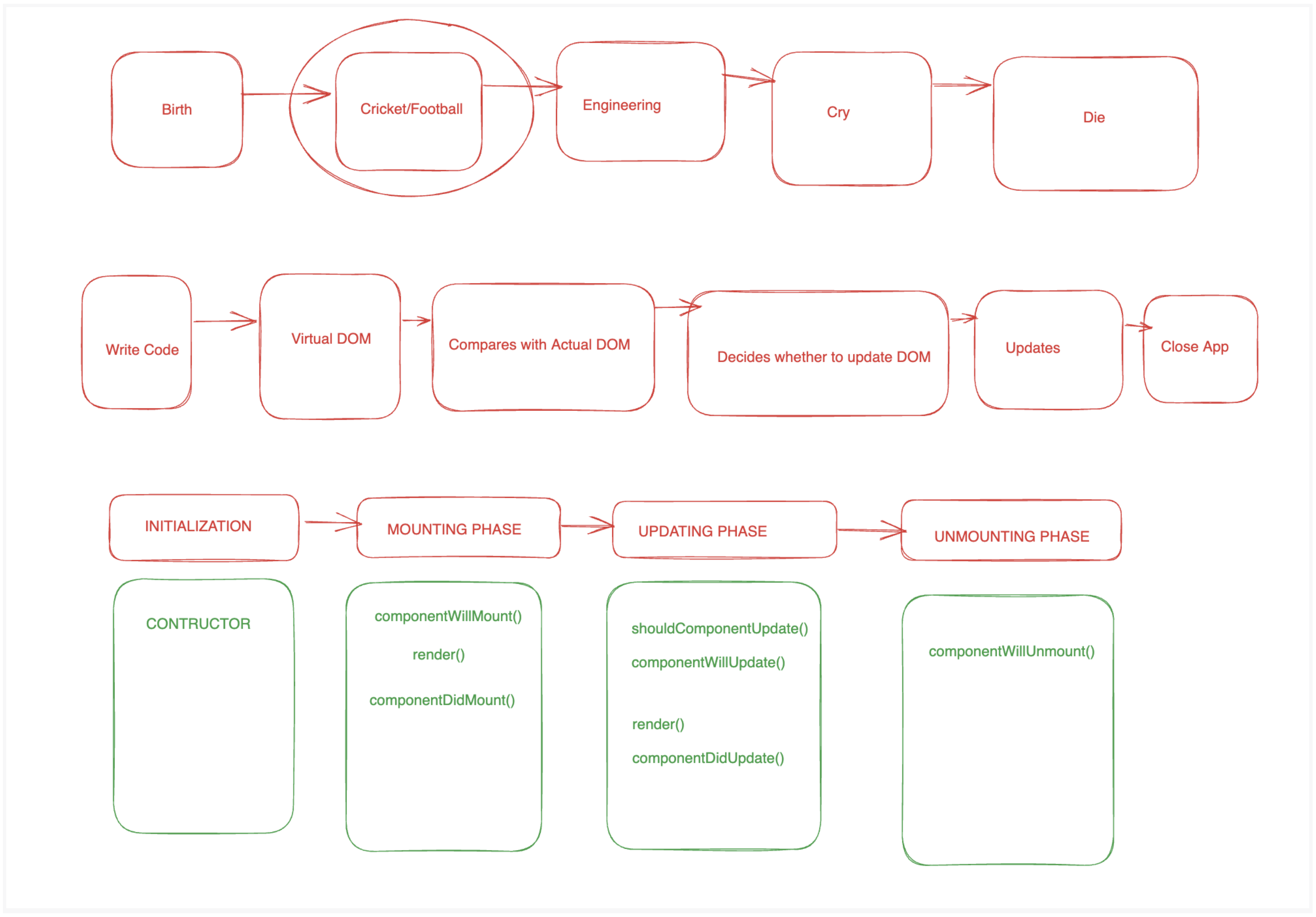
```
class MyComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    // Only update if the count is odd
    return nextState.count % 2 === 1;
  }

  render() {
    return <div>{this.state.count}</div>;
  }
}
```

5. componentWillUnmount(): The componentWillUnmount method is called right before the component is unmounted and destroyed. It is used to clean up any resources or subscriptions created in componentDidMount. Here's an example:

```
class MyComponent extends React.Component {
  componentDidMount() {
    this.timerId = setInterval(() => {
      console.log('Tick');
    }, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.timerId);
  }

  render() {
    return <div>Component with timer</div>;
  }
}
```

Combining all the above in a single code we get :

```
class LifecycleExample extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    console.log('Constructor');
  }

  static getDerivedStateFromProps(props, state) {
    console.log('getDerivedStateFromProps');
    return null;
  }

  componentDidMount() {
    console.log('componentDidMount');
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log('shouldComponentUpdate');
    return true;
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log('getSnapshotBeforeUpdate');
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    console.log('componentDidUpdate');
  }

  componentWillUnmount() {
    console.log('componentWillUnmount');
  }

  incrementCount = () => {
    this.setState(prevState => ({
      count: prevState.count + 1
    }));
  };

  render() {
    console.log('render');
    return (
      <div>
        <h1>Lifecycle Methods Example</h1>
        <p>Count: {this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```


Birth
Cricket/Football
Engineering
Cry
Die
Write Code
Virtual DOM
Compares with Actual DOM
Decides whether to update DOM
Updates
Close App
INITIALIZATION
MOUNTING PHASE
UPDATING PHASE
UNMOUNTING PHASE
CONTRUCTOR
componentWillMount()
render()
componentDidMount()
shouldComponentUpdate()
componentWillUpdate()
render()
componentDidUpdate()
componentWillUnmount()